A Modular Meta-model for Security Solutions

Laurens Sion
imec-DistriNet
KU Leuven
laurens.sion@cs.kuleuven.be

Koen Yskout
imec-DistriNet
KU Leuven
koen.yskout@cs.kuleuven.be

Riccardo Scandariato
Software Engineering Division
Chalmers & Göteborg University
riccardo.scandariato@cse.gu.se

Wouter Joosen
imec-DistriNet
KU Leuven
wouter.joosen@cs.kuleuven.be

ABSTRACT

Designing a secure software system requires the ability to represent and reason about a wide variety of security concerns. Existing modelling representations lack a comprehensive set of security building blocks or lack support for composition or refinement of the design under consideration. We propose a new modular meta-model for representing these security designs. This model supports both composition for more complex solutions and representing different levels of abstraction to model the underlying details. This meta-model can subsequently be used for the construction of security solutions, supporting a wide range of mechanisms on a wide variety of abstraction levels, thereby providing a foundation for the security-by-design approach.

KEYWORDS

security, design, meta-model

ACM Reference format:
Laurens Sion, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2017. A Modular Meta-model for Security Solutions. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 5 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079393

1 INTRODUCTION

With society's growing reliance on technology, software systems have become increasingly complex and their security has become increasingly important to get right. The importance of properly designed software is already recognized in software engineering for a very long time [7]. Its equal importance with regard to security is also gaining more attention recently with security extensions for UML such as UMLsec [9] and several security design pattern catalogues [5, 13].

In the software architecture domain, a widespread stance is that security-relevant information of a software system should be captured in a *view*, which is composed of one or more *models* [3, 8]. Such a (system-specific) view is an instance of a security *viewpoint*, which prescribes the *model kinds* to use. A model kind specifies, among others, the language and notation for the models, as well as analysis methods and operations, and is typically backed by a meta-model. The availability of a precise security view for a system opens the door for extensive analysis activities to verify that a design meets its security requirements. Besides modelling the security of a concrete system, such a view can also serve as the basis for describing security patterns and solutions, enabling a precise, reusable, and unambiguous dissemination of them. To enable these uses, a suitable underlying meta-model is required.

Unfortunately, for the security domain, such a generic meta-model currently does not exist, causing the creation of a security view to remain a difficult task. For example, the survey of van den Berghe et al. [15] illustrated that many representations lack a comprehensive set of elements and only focus a single security concern. Existing catalogues of security patterns further corroborate this, by using custom or ad-hoc representations (e.g., variants of UML) that are not always well-suited to express the solutions [5, 6, 12, 13]. Such ambiguous ad-hoc representations can cause misinterpretations or faulty instantiations of these security patterns and prohibits easy comparison of multiply pattern alternatives.

A challenge when creating a security meta-model is that it needs to support the creation of an integrated, comprehensive, and simultaneously in-depth view on software security. Issues in secure API development illustrate the need for such a view. An example of this is the XOR-To-Null-Key attack in a bank terminal API, discussed by Anderson [1], which illustrates how the combination of secure instructions can lead to insecurity. The meta-model should therefore provide modular support for a comprehensive set of security building blocks and solutions, enable the modelling of complex security-sensitive systems by composing partial solutions, allow to refine the model to an arbitrary level of detail where necessary, and enable reasoning about security attacks.

In this paper, we propose such a modular meta-model for security. It has been designed with the explicit intent of enabling composition and allowing refinement, and to serve as the foundation of a *security-by-design* approach. Since space constraints prevent us from providing and discussing all aspects and subtleties of the proposed meta-model, we necessarily had to make a selection of the concepts to present in this paper. But this selection does serve as a good illustration of the expressiveness and modular support for representing models in various abstraction levels.

Programming '17, Brussels, Belgium

978-1-4503-4836-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079393

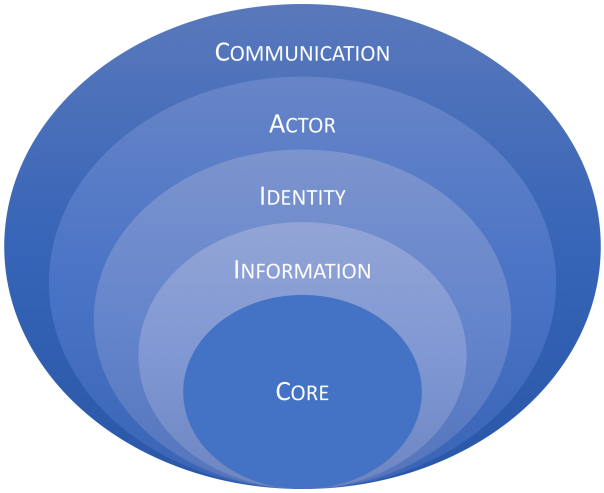

Figure 1: Structure of our meta-model

2 PROPOSED META-MODEL

In this section the proposed meta-model is elaborated upon. This model is designed to adhere to the following principles.

Comprehensiveness. The meta-model should support expressing a wide range of varying security solutions and attacks. It should not limit the designer to a predefined fixed set of security concerns.

Modularity. The model should support expressing complex and large systems by composing elements into larger solutions. This is enabled in the proposed meta-model by using the composite pattern [7]. Besides constructing more complex solutions, the model should also be modular in supporting representations at varying levels of abstraction. In other words, the model should not force or limit the designer to work on a single level of abstraction. Since the details of a specific security solution are important to its correctness, the meta-model should support the inclusion of all the lower-level details into the model.

Figure 1 shows the structure of the proposed meta-model. This structure of the meta-model is layered, of which each layer uses and builds on top of the previous layers. The rest of this section presents each of these layers in turn, enriched with examples, including security mechanisms and attacks.

2.1 Core Layer

The CORE meta-model layer contains a very small set of meta-model classes, encapsulating shared properties of all meta-model classes such as a name attribute. This layer is not very interesting by itself.

2.2 Information Layer

The INFORMATION layer is an essential layer of the security meta-model. Information is the main asset used in many software systems and processing information is one of the core responsibilities of software systems. Therefore, securing these software systems requires placing additional constraints on the information processing, i.e., controlling what is processed when, how, and by whom.

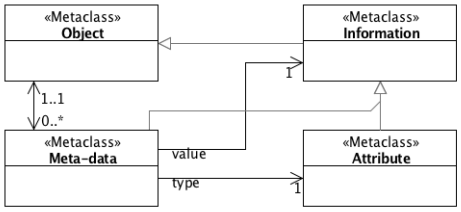

Figure 2: Information layer

The INFORMATION layer contains classes to generically represent information and its meta-data, and is (partially) displayed in Figure 2. *Object* is a generic representation for something which can have meta-data associated with it. A *Meta-data* instance is used to link the actual meta-data to the object, by linking to the type of meta-data (*Attribute*) and the value (*Information*). Both *Meta-data* and *Attribute* are themselves specializations of *Information*, which in turn is a specialization of *Object*, so they can all have meta-data associated with them as well. This meta-model layer also allows the construction of more complex *Information*-structures such as, for example, lists (associating each entry with its index via meta-data).

Example: Meta-data and nested meta-data. As an illustration of how *Meta-data* is associated with *Objects*, consider the example of a person (*Object*) with an age (*Attribute*) of 50 years (*Information*), as shown in Figure 3. Note how the *Information* object that represents the age value has its own *Meta-data*, which distinguishes the value (50) from the unit (years).

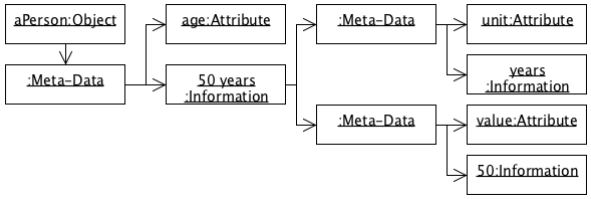

Figure 3: Example of meta-data and nested meta-data

Example: Cryptographic key. The classes from the INFORMATION layer can also be used to represent cryptographic keys. In this representation, there is a *cryptoKey* object which has the meta-data associated with it for the key's properties. This can include elements such as the key size, the algorithm, an identifier, etc.

2.3 Identity Layer

A second important layer for modelling security is the IDENTITY layer. Identities are required for distinguishing between different entities interacting with a system, which can be human users as well as other systems. When devices and services can have more than one user, it becomes more important to be able to distinguish between them. The need for identities and verifying these identities is also apparent in the real world, where many interactions with

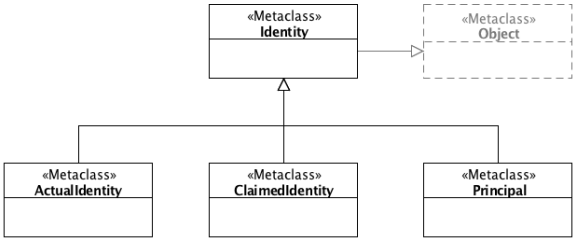

Figure 4: Identity layer

companies or government agencies require some form of identity information to be verified.

The IDENTITY layer contains the classes for representing identities (Figure 4). There is one root class *Identity*, and three specializations of this class. An instance of any of the *Identity* classes contains attributes with information about the entity to which it belongs. For example, for a human user, this could include the user's real name, address, usernames and passwords on various websites, or iris pattern. For a server, this could be the IP address, the physical location, hostname, or security certificate, for instance. *ActualIdentity* is used to represent one unique identity of an entity in the model, representing the 'true' identity of the entity. The *ClaimedIdentity* is used to communicate identity information to other entities. To this end, it may or may not contain information of *ActualIdentity* (e.g., consider a stolen identity document). The third specialization, *Principal*, represents how knowledge about some identity is stored by someone else. An example of this is storing identity information in a user database.

Example: Authentication. In this example, we show how the model can be used to express authentication and potential attacks on authentication mechanisms.

Authentication consists of two steps: *identification* and *verification*. Firstly, the entity that needs to be authenticated identifies itself by providing identity information in a *ClaimedIdentity*. Next, in the verification step, the authenticator needs to verify the information in the *ClaimedIdentity* against the known information (i.e., the *Principal*). We illustrate an authentication procedure in detail with the following list of steps. The identities[1] in these steps use the following super- and subscripts: $X^{@storedAt}_{subject,creator/modifier}$. The *creator/modifier* can be omitted if it is the same as the subject. Note that these subscripts are only used in the example to distinguish between different *instances* of the *Identity* class. They do not reflect attributes or associations contained in the meta-model itself, although some of the information that they convey could be associated with the identities as meta-data.

(1) The user (AI_{user}) wants to visit a website ($AI_{website}$).
(2) The user types the URL in the browser as (s)he remembers it ($P^{@user}_{website,user}$)
(3) The user is (hopefully) connected to the correct website ($AI_{website}$), but the user can only verify this using the information received over the connection ($CI_{website,x}$, where x can be the website or an attacker)
(4) The user provides her/his credentials (CI_{user}) to the website.
(5) The system receives login information (CI_{user}). Depending on security mechanisms in place, the system may receive modified ($CI_{user,attacker}$) information (e.g., attacker in the middle).
(6) The system checks if the information (CI_{user}) received corresponds with the known account details ($P^{@website}_{user,website}$).

That last step reveals how authentication can be modelled. Successful authentication occurs when the system can confirm that the *ClaimedIdentity* corresponds with the known *Principal*: $CI_{user,x} \sim P^{@server}_{user,server}$. Failed authentication is the opposite, i.e. $CI_{user,x} \nsim P^{@server}_{user,server}$.

Spoofing also fits in this model. In case of a successful spoofing attack, an adversary can construct a *ClaimedIdentity* which closely resembles the user's own one ($CI_{user} \approx CI_{user,attacker}$) and that matches with the *Principal* ($P^{@website}_{user,website}$) known by the website. The problem is of course that the *ClaimedIdentity* does not correspond with the *ActualIdentity*: $CI_{user} \sim AI_{user}$ but $CI_{user,attacker} \nsim AI_{user}$.

Similarly, spoofing of the server can also be represented. In this case, the attacker fabricates the *ClaimedIdentity* of the server $CI_{server,attacker} \approx CI_{server}$, thereby tricking the user into thinking (s)he is communicating with the server (AI_{server}) while in reality communicating with the attacker ($AI_{attacker}$).

2.4 Actor Layer

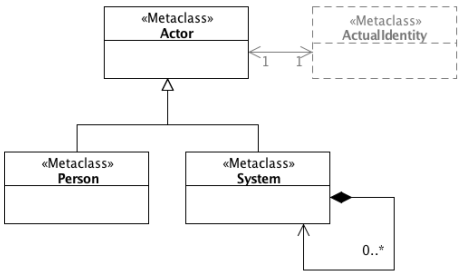

Figure 5: Actor layer

The classes from the ACTOR meta-model layer are used to represent any entity in the system that performs actions (e.g., running code, communicating). Figure 5 shows the actor classes. This layer contains the *Actor* class, which is specialized into a *Person* and a *System* class. The *System* class supports recursive decomposition, so more complex *System* structures can be modelled, and the system can be represented at multiple levels of abstractions by decomposing it into the underlying and interacting elements. For example, a server *Actor* can be decomposed into a web server and operating system actor when this level of detail is required. Additionally, every *Actor* is also associated with an *ActualIdentity* representing the 'true' identity of the *Actor*.

[1] AI: *ActualIdentity*, CI: *ClaimedIdentity*, P: *Principal*

Example: Phishing. Phishing also includes the concepts from the IDENTITY-layer. In a phishing attack, an adversary sends a *ClaimedIdentity* ($CI_{website,phisher}$) to the end-user, which contains attributes that look very similar to those of the *ActualIdentity* of the *Actor* being spoofed. E.g., a *ClaimedIdentity* with the following URL `https://www.paypaI.com` is sent to the user. In this example, the inaccurate manual verification step of the human end-user (*Person*) is exploited or the fact the UI of the browser (*System*) of the end-user does not make a visual distinction between an lowercase ℓ and a capital I. To counter this threat, additional security mechanisms can be introduced which are modelled as follows. The *ClaimedIdentity* of the phisher will contain the misleading URL $CI_{website,phisher}\{url(..)\}$. To make such a successful attack more difficult, the *System* will include additional information which is hard to fabricate (e.g., certificate with organizational information): $CI_{website}\{url(..); \text{certificate}(url, OrganizationName)\}$. This information is used by the browser (*System*) and is displayed to the user (*Person*) to avoid confusion. This happens by, for example, adding the green address bar for extended validation certificates and displaying the organization name next to the website address.

2.5 Communication Layer

This layer links to the ACTOR, IDENTITY and INFORMATION layers discussed above. The COMMUNICATION-layer provides classes for modelling communication of varying complexity between *Actor*s, and this on varying levels of abstraction. There are three main groups of communication modelling elements: (i) representing the communication channels themselves (**Channel**, **Port**, **Interface**), (ii) representing how communication happens over those channels (**InstructionSet**, **TransmissionContext**, **Transmission**, **Communication**, **Session**), and (iii) representing how communication channels can be decomposed into the underlying channels on lower levels of abstraction (**InterfaceConcretization**, **ChannelConcretization**, **InterfaceLink**, **PortLink**). The classes for modelling channels (i) and their concretization (iii) are displayed in Figure 6, while the classes for modelling the communication over these channels (ii) are displayed in Figure 7.

For the representation of communication channels (i), three elements are used. *Channel* represents the communication link between two interacting *Actor*s. To interact with the communication *Channel*, actors can *control* or *observe* an *Interface* (or both), which is tied to a channel via a *Port*.

To communicate *Information* over this channel, each *Interface* has an *InstructionSet* associated with it, which expresses how the *Interface* can be used to send *Information* over the *Channel*. Each *Transmission* over the channel links a *TransmissionContext* at the sending side with one at the receiving side. The *TransmissionContext* links to the actual message that is sent or received, expressed in the *InstructionSet* of respectively the sending or receiving *Interface*. The separate representation of both the sent and received message enables modelling modifications of transmissions (e.g., by an intermediary such as a router or an attacker). Above the *Transmission* and *TransmissionContext* are *Communication* and *Session*, which enable modelling more complex communications such as the run of a protocol.

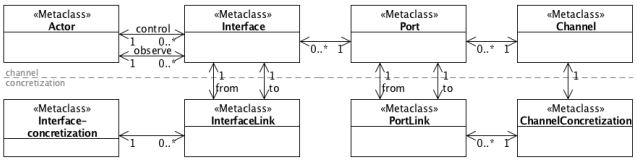

Figure 6: Communication layer: Communication channel and concretization elements

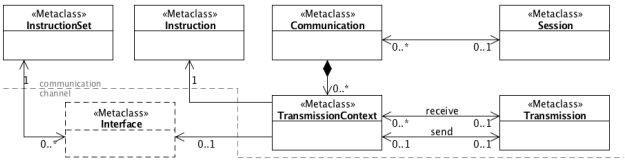

Figure 7: Communication layer: Communication transmission elements

Finally, the COMMUNICATION-layer also contains classes for modelling the concrete details underlying a *Channel* (iii). This is done by using a *ChannelConcretization*. The *ChannelConcretization* links the *Port*s of the *Channel* being refined, to the *Port*s of the underlying *Channel*s. This expresses how the combination of the underlying *Channel*s forms the original *Channel*. Similar to the concretization of *Channel*s, the *Interface*s are concretized as well. For linking the *Interface*s, an *InterfaceConcretization* is added which links the high-level *Interface* with the lower-level *Interface*. By making this link explicit, the *InterfaceConcretization* can also be associated with the algorithms necessary for translating communication in the high-level *InstructionSet* to communication in the low-level *InstructionSet*, there by making explicit how high-level communication is transformed to low-level messages. This way of modelling enables the inclusion of actual protocols in the model, they are represented as a combination of algorithms, which are included explicitly in the model at both ends of the communication.

Example: Man-in-the-Middle. An example of a type of attack that can be modelled using the communication classes is a man-in-the-middle attack. The elements for representing *Channel*s can be used to explicitly model what happens on the channels underneath it. For example, the entire TCP/IP stack can be modelled by breaking down the channel in each layer into the underlying elements. This enables modelling man-in-the-middle attacks, but also any *Actor*s in between communicating *Actor*s that perform some functionality. An example of this is when Network Address Translation is used.

Especially when conducting security analyses, taking the complete picture into account, including all underlying layers, is necessary to make proper statements with regard to the security of the overall solution. This is the security version of the end-to-end argument, which has already been discussed extensively before [11].

Example: Side-channel. The proposed communication classes can also be used to model side-channel attacks. All communication between *Actor*s happens over channels, and any interaction from an *Actor* with a *Channel* happens through an *Interface*. Therefore, a side-channel can be modelled as an additional *Interface* which

allows an adversary to observe the communication over the *Channel*. This *Interface* can have a completely different *InstructionSet*. E.g., measuring the current in a wire.

Example: Protections offered by lower layers. Finally, the modelling of *ChannelConcretization*s enables the explicit inclusion of dependencies on mechanisms applied in lower-level *Channel*s. Consider, for example, authentication via a username and password combination. This security mechanism relies on properly verifying these user credentials at the receiving end of the channel. But, in addition to this security check, the mechanism also strongly relies on the confidentiality of these credentials. To realize this protection, websites often rely on a SSL/TLS channel below the HTTP channel. Such a critical requirement on another underlying security mechanism can be made explicit in this model.

3 RELATED WORK

Van den Berghe et al. [15] conducted a systematic literature survey on representations for security. They identified a major gap in the support for several different security concerns, with several representations focusing on only a single concern. Nguyen et al. [10] did survey on model-driven development for secure systems. The results for model-driven security confirm the above results, with very few approaches supporting multiple security concerns. Additionally, interactions between security concerns are often overlooked. Uzunov et al. [14] conducted a more broadly scoped survey on approaches combining security and software engineering. They conclude there is no ideal methodology that satisfied all their criteria. Their survey also confirms the lack of support for a wide range of security solutions.

Because of the large amount of different security representations, we limit ourselves to a discussion of a few important examples of existing representations. UMLsec [9] offers UML profile for including security information and makes extensive use of stereotypes and tagged values to represent this information. It is geared towards automated verification, and thus focusses on a lower level of abstraction. Another representation is SecureUML [2] for model-driven security. In SecureUML a combination is made of modelling languages (e.g., UML) and security language (e.g., an RBAC representation) to formalize access control requirements and generate the necessary infrastructure. Illustrating a scoped focus on a specific security concern.

Finally, Bau and Mitchell [4] elaborated on the need for a uniform conceptual framework for expressing systems, threats, and security properties. The framework still targets a low level of abstraction, as it is used for protocol analysis.

4 FUTURE WORK

The parts of the meta-model presented above form the structure of the system to which security mechanisms are to be applied. In this section we list a selection of future extensions to the meta-model.

The current meta-model classes offer a solid base for expressing a wide range of security mechanisms in its models. In the next steps, we want to leverage this model for expressing common security mechanisms in a reusable way, including the links of these mechanisms to the different system model elements that they use or interact with, the explicit dependencies on other security mechanisms, and assumptions of these mechanisms on other elements of the model (e.g., underlying confidential channel).

Besides, the extension mechanisms for the representation of security mechanisms and their interaction with the system model elements, there are also opportunities for providing a library of system representations that can be reused. Examples of this are: the standard TCP/IP network stack, the HTTP and SSL/TLS combination, and common platforms or frameworks. Common systems can then be readily reused and do not need to be remodelled.

5 CONCLUSION

In order to provide a comprehensive security model, support is needed for (i) an extensive set of building blocks for constructing security solutions, (ii) support for composing and decomposing more complex security solutions, and support for refinement by representing the details and inner workings of a solution on a lower abstraction level. Currently, many representations lack support for multiple security mechanisms. Additionally, existing solutions have only limited support for composition and refinement.

In this paper, we introduced a meta-model which supports modelling a variety of security mechanisms and attacks, and in addition supports composition of more complex solutions and refinement into lower levels of abstractions for working out the details. The proposal is to serve as a stepping stone for a more detailed and extensive security model in support of a *security-by-design* approach.

REFERENCES

[1] Ross Anderson. 2008. *Security Engineering* (2nd ed.). Wiley.
[2] David Basin, Jürgen Doser, and Torsten Lodderstedt. 2006. Model driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 1 (2006).
[3] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.
[4] Jason Bau and John C Mitchell. 2011. Security modeling and analysis. *Security & Privacy, IEEE* 9, 3 (2011), 18–25.
[5] Eduardo Fernandez-Buglioni. 2013. *Security patterns in practice: designing secure architectures using software patterns.* John Wiley & Sons.
[6] Eduardo Fernández-Medina, Jan Jurjens, Juan Trujillo, and Sushil Jajodia. 2009. Model-Driven Development for secure information systems. *Information and Software Technology* 51, 5 (may 2009).
[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design patterns: elements of reusable object-oriented software.* Pearson Education.
[8] ISO/IEC/IEEE. 2011. Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E)* 1 (2011), 1–46.
[9] Jan Jürjens. 2005. *Secure Systems Development with UML.* Springer Berlin Heidelberg.
[10] Phu H. Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. 2015. An extensive systematic review on the Model-Driven Development of secure systems. *Information and Software Technology* 68 (2015).
[11] J H Saltzer, D P Reed, and D D Clark. 1984. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (nov 1984).
[12] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns.* Wiley.
[13] Christopher Steel, Ramesh Nagappan, and Ray Lai. 2005. *Core Security Patterns: Best Pratices and Strategies for J2EE, Web Services, and Identity Management.* Prentice Hall Ptr.
[14] Anton V. Uzunov, Eduardo B. Fernandez, and Katrina Falkner. 2012. Engineering Security into Distributed Systems: A Survey of Methodologies. *Journal of Universal Computer Science* 18, 20 (2012).
[15] Alexander van den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. 2015. Design notations for secure software: a systematic literature review. *Software & Systems Modeling* (2015).